THE ISLAMIC UNIVERSITY – FACULTY OF ENGINEERING

COMPUTER ENGINEERING DEPARTMENT

LCOM 3010 – Introduction to Computer Architecture Lab

Lab #1
INTRODUCTION OF MIPS

BY

Eng: Rasha Ghazy Sammour

Submitted for

Dr. Mohammed Mikki

March 31, 2021

Introduction of MIPS instruction by Assembly Language

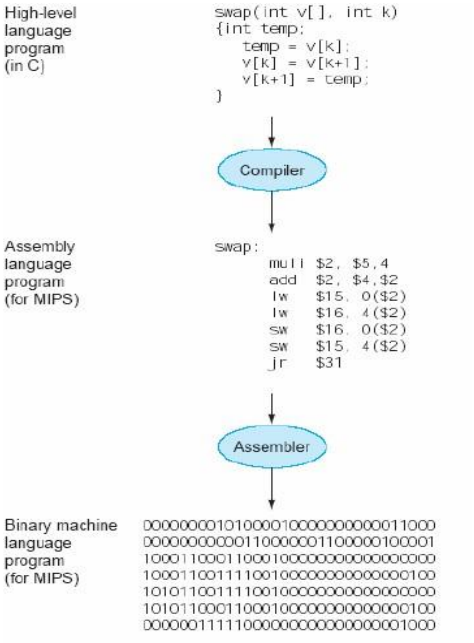
Objective:

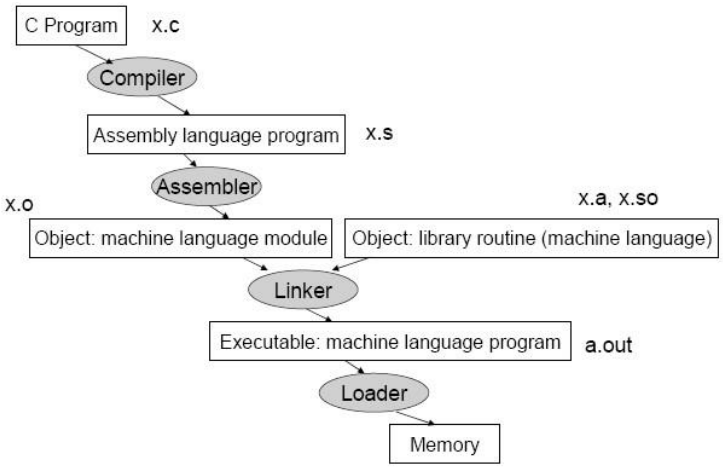
1. Write a simple program in assembler language to implement a high level program segment.
2. Design simple assembly language programs that make appropriate use of a registers and memory.

Theory: Assembly Language:

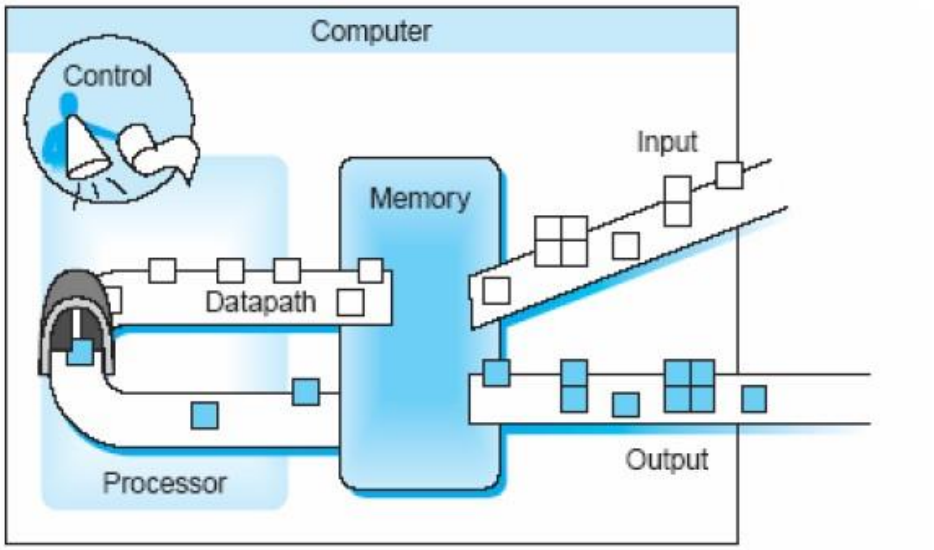
Below Your Program

High level languages:
1)Allow to think in a more natural way
2) Improve programmer productivity
3) Allows programs to be independent of the computer





Computer Organization:



The five classic components: 1) Input, 2) Output, 3) Memory, 4) Datapath 5) Control

A Basic MIPS Instructions:

C code: a = b + c ;

Assembly code: (human-friendly machine instructions)

```
add   a, b, c     #  a is the sum of b and c
```

Machine code: (hardware-friendly machine instructions)

```
00000010001100100100000000100000
```

Translate the following C code into assembly code:

```
a = b + c + d + e;
```

Examples:

C code a = b + c + d + e;
translates into the following assembly code:

```
add  a, b, c          add  a, b, c
add  a, a, d    or    add  f, d, e
add  a, a, e          add  a, a, f
```

- Instructions are simple: fixed number of operands (unlike C)
- A single line of C code is converted into multiple lines of assembly code
- Some sequences are better than others… the second sequence needs one more (temporary) variable f

Subtract Example:

C code f = (g + h) – (i + j);
translates into the following assembly code:

```
add  t0, g, h          add  f, g, h
add  t1,  i, j   or    sub  f, f, i
sub  f,   t0, t1       sub  f, f, j
```

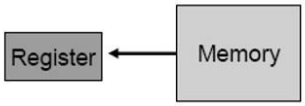
Operands:

- In C, each "variable" is a location in memory
- In hardware, each memory access is expensive – if variable *a* is accessed repeatedly, it helps to bring the variable into an on-chip scratchpad and operate on the scratchpad (registers)
- To simplify the instructions, we require that each instruction (add, sub) only operate on registers
- Note: the number of operands (variables) in a C program is very large; the number of operands in assembly is fixed… there can be only so many scratchpad registers

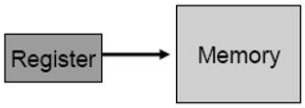
Memory Operands:

- Values must be fetched from memory before (add and sub) instructions can operate on them

Load word
lw $t0, memory-address



Store word
sw $t0, memory-address



Immediate Operands:

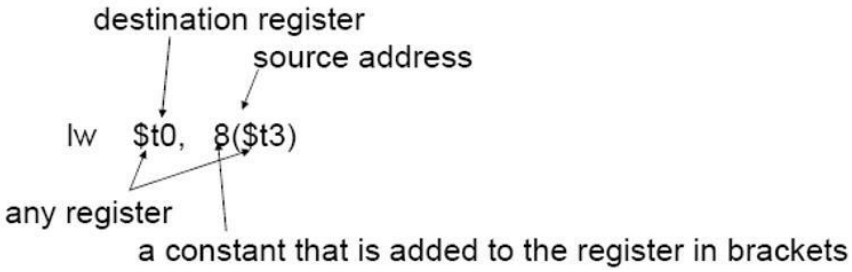
- An instruction may require a constant as input

- An immediate instruction uses a constant number as one of the inputs (instead of a register operand)

```
addi   $s0, $zero, 1000   # the program has base address
                          #  1000 and this is saved in $s0
                          # $zero is a register that always
                          # equals zero
addi   $s1, $s0, 0        # this is the address of variable a
addi   $s2, $s0, 4        # this is the address of variable b
addi   $s3, $s0, 8        # this is the address of variable c
addi   $s4, $s0, 12       # this is the address of variable d[0]
```

Memory Instructions Format:

- The format of a load instruction:



Example:

Convert to assembly:

C code: d[3] = d[2] + a;

Assembly: # addi instructions as before

```
lw   $t0, 8($s4)   # d[2] is brought into $t0
lw   $t1, 0($s1)   #  a  is brought into $t1
add  $t0, $t0, $t1  # the sum is in $t0
sw   $t0, 12($s4)  # $t0 is stored into d[3]
```

Assembly version of the code continues to expand!

Logical Operands:

Logical ops	C operators	Java operators	MIPS instr
Shift Left	<<	<<	sll
Shift Right	>>	>>>	srl
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR	|	|	or, ori
Bit-by-bit NOT	~	~	nor

```
or   $t0, $t1, $t2
ori  $t0, $t1, 255
sll  $t0, $t1, 4
and $t0, $t1, $t2
```

Control Instructions:

- Conditional branch: Jump to instruction L1 if register1 equals register2: beq register1, register2, L1
  Similarly, bne and slt (set-on-less-than)

- Unconditional branch:
  j L1
  jr $s0

Convert to assembly:

```
if (i == j)                    bne  $s3, $s4, Else
   f = g+h;                    add  $s0, $s1, $s2
else                           j    Exit
   f = g-h;              Else: sub  $s0, $s1, $s2
                         Exit:
```

Examples:

Convert to assembly:

```
while (save[i] == k)
   i += 1;
```

i and k are in $s3 and $s5 and base of array save[] is in $s6

```
Loop: sll   $t1, $s3, 2
      add   $t1, $t1, $s6
      lw    $t0, 0($t1)
      bne   $t0, $s5, Exit
      addi  $s3, $s3, 1
      j     Loop
Exit:
```

Example:

```
if (save[i] >= k)
    Code;
```

Assume save[i] is stored in $t1 and k is stored in $t2

```
    slt   $t0, $t1, $t2
    bne   $t0, $zero, Exit
    Code
Exit:
```

Example2:

```
if (save[i] < k)
    Code;
Assume save[i] is stored in $t1 and k is stored in $t2
        slt   $t0, $t1, $t2
        beq  $t0, $zero, Exit
        Code
Exit:
```

Example3:

```
if (save[i] > k)
    Code;
Assume save[i] is stored in $t1 and k is stored in $t2
        slt   $t0, $t1, $t2
        bne  $t0, $zero, Exit
        beq  $t1, $t2, Exit
        Code
Exit:
```

Try it by yourself:

```
if (save[i] <= k)
    Code;
Assume save[i] is stored in $t1 and k is stored in $t2
```

Registers:

• The 32 MIPS registers are partitioned as follows:

- Register 0 : $zero always stores the constant 0
- Regs 2-3 : $v0, $v1 return values of a procedure
- Regs 4-7 : $a0-$a3 input arguments to a procedure
- Regs 8-15 : $t0-$t7 temporaries
- Regs 16-23: $s0-$s7 variables
- Regs 24-25: $t8-$t9 more temporaries
- Reg 28 : $gp global pointer
- Reg 29 : $sp stack pointer
- Reg 30 : $fp frame pointer
- Reg 31 : $ra return address

Jump and Link:

- A special register (storage not part of the register file) maintains the address of the instruction currently being executed – this is the *program counter* (PC)

- The procedure call is executed by invoking the jump-and-link (jal) instruction – the current PC (actually, PC+4) is saved in the register $ra and we jump to the procedure's address (the PC is accordingly set to this address)

```
jal   NewProcedureAddress
```

- Since jal may over-write a relevant value in $ra, it must be saved somewhere (in memory?) before invoking the jal instruction

The Swap Procedure:

- Register allocation: $a0 and $a1 for the two arguments, $t0 for the temp variable – no need for saves and restores as we're not using $s0-$s7 and this is a leaf procedure (won't need to re-use $a0 and $a1)

```
swap:  sll   $t1, $a1, 2
       add   $t1, $a0, $t1
       lw    $t0, 0($t1)
       lw    $t2, 4($t1)
       sw    $t2, 0($t1)
       sw    $t0, 4($t1)
       jr    $ra
```

```
void swap (int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

Summary:

name		Format					syntax	meaning
	Op(6)	rs(5)	rt(5)	rd(5)	shamt(5)	func(6)		
add	0				0	32	add rd,rs,rt	rd = rs + rt
sub	0				0	34	sub rd,rs,rt	rd = rs - rt
and	0				0	36	and rd,rs,rt	rd = rs AND rt
or	0				0	37	or rd,rs,rt	rd = rs OR rt
sll	0	0				0	sll rd,rt,shamt	rd = logical shift rt left shamt bits
srl	0	0				2	srl rd,rt,shamt	rd = logical shift rt right shamt bits
slt	0				0	42	slt rd,rs,rt	if rs<rt set rd=1 else rd=0
jr	0		0	0	0	8	jr rs	PC=rs
	Op(6)	rs(5)	rt(5)	immediate(16)				
addi	8						addi rt,rs,immed	rt = rs + immed
andi	12						andi rt,rs,immed	rt = rs AND immed
ori	13						ori rt,rs,immed	rt = rs OR immed
lw	35						lw rt,immed(rs)	rt = MEMORY[rs+immed]
sw	43						sw rt,immed(rs)	MEMORY[rs+immed] = rt
lui	15						lui rt,immed	rt = immed shifted left 16 bits
beq	4						beq rs,rt,label	branch if equal
bne	5						bne rs,rt,label	branch if not equal
	Op(6)	target address(26)						
j	2						j label	jump
jal	3						jal label	jump and link

Assignment Statements:

1. Show the single MIPS instruction for this C statement: b = 25 | a;
2. Convert the following C fragment to equivalent MIPS assembly language. Assume that the variables a and b are assigned to registers $s0 and $s1 respectively. Assume that the base address of the array D is in register $s2.

```
While (a < 10) {
    D[a] = b + a;
    a += 1;
}
```